86-0056

ROSIE® A Programming Environment for Expert Systems

Henry A. Sowizral and James R. Kipps

Rand

The research described in this report was sponsored by the Defense Advanced Research Projects Agency under ARPA Order No. 3460-9, Contract No. MDA903-82-C-0061, Information Processing Techniques.

Library of Congress Cataloging in Publication Data

Sowizral, Henry A., 1955-
ROSIE : a programming environment for expert systems.

"Prepared for the Defense Advanced Research Projects Agency."
"October 1985."
"R-3246-ARPA."
Bibliography: p.
1. Expert systems (Computer science) 2. ROSIE (Computer system) I. Kipps, James R. (James Randall), 1960- . II. United States. Defense Advanced Research Projects Agency. III. Rand Corporation. IV. Title.
QA76.76.E95S69 1985 006.3'3 85-20494
ISBN 0-8330-0676-2

Published by The Rand Corporation

R-3246-ARPA

ROSIE® A Programming Environment for Expert Systems

Henry A. Sowizral and James R. Kipps

October 1985

Prepared for the
Defense Advanced Research
Projects Agency

Rand
1700 MAIN STREET
P.O. BOX 2138
SANTA MONICA, CA 90406-2138

PREFACE

The ROSIE[1] project is an ongoing research effort concerned with expert system language design and development. This report provides an overview of the ROSIE language, a description of its new concepts and features, an evaluation of its successes and failures to date, and an indication of future research directions. The version described in this report is ROSIE 2.5, which runs under Interlisp-D (on Xerox SIP 1100 series processors) and VAX-Interlisp (on VAX computers). ROSIE 2.5 is also currently being ported to PSL.

This report is not a reference manual. The primary documentation on ROSIE is contained in Hayes-Roth et al., *Rationale and Motivation for ROSIE,* The Rand Corporation, N-1648-ARPA, November 1981; Fain et al., *The ROSIE Language Reference Manual,* The Rand Corporation, N-1647-ARPA, December 1981; and Fain et al., *Programming in ROSIE: An Introduction by Means of Examples,* The Rand Corporation, N-1646-ARPA, February 1982. These documents describe the earlier ROSIE 1.0 version. A new ROSIE 2.5 reference manual is currently being written.

This work was supported by the Defense Advanced Research Projects Agency (Information Processing Techniques Office) under contract MDA-903-82-C-0061.

[1]ROSIE is a trademark of The Rand Corporation.

SUMMARY

ROSIE is an English-like programming language that has evolved over the years into a highly readable, expressive, and powerful tool for building expert systems. Because the language mirrors English, it also serves as a medium of interaction between the knowledge engineer (the computer expert who creates the expert system) and the domain expert (the person whose expertise the system reflects).

This report describes the ROSIE language, emphasizing recent changes and additions. The changes have been made mainly to the internals of ROSIE. They make the language far more perspicuous; they simplify its structure and make it more modular. Additions to the language include *meta-elements*, *shared databases*, and *demons*.

Meta-elements are ROSIE elements that capture specific linguistic structures. The three meta-elements described in this report are *propositions* (basic sentences), *intentional descriptions* (methods for accessing elements in the database), and *intentional actions* (suspended procedure calls).

Shared databases provide ROSIE with a facility for coordinating interactions among multiple experts without introducing significant changes to the language. Shared databases act just like databases; however, several ROSIEs can access and modify the shared database concurrently.

Demons, programs that awaken on specific conditions, provide ROSIE with additional power. They can guard databases to prevent inconsistencies; they can mimic a frame-like programming style; and, together with shared databases, they can provide an effective mechanism for controlling communication between expert systems.

This report presents an overview of the ROSIE language and describes its new facilities. The advantages and disadvantages of ROSIE are discussed, and a look is taken at future directions.

ACKNOWLEDGMENTS

We would like to thank the people involved with the current version of ROSIE, especially Larry Baer, Jill Fain, Bruce Florman, and Jed Marti. We would also like to thank Robert Anderson and Marietta Gillogly for their help in developing this report. James Dewar and Geneva Henry provided insightful and cogent reviews. Their suggestions substantially improved the report. But especially, we would like to thank Philip Klahr for repeatedly reading and commenting on earlier drafts. We alone take responsibility for any errors or inaccuracies that remain.

CONTENTS

I. INTRODUCTION

WHAT IS ROSIE?

ROSIE (Rule-Oriented System for Implementing Expertise) is a programming environment for artificial intelligence (AI) applications. It provides particular support for designing *expert systems*, systems that embody knowledge of a domain and operate using that knowledge.

ROSIE uses a near-English syntax for representing facts and rules. A person who is not familiar with programming languages can pick up a ROSIE program, read it, and understand it, almost as if it were English. The design of the ROSIE language rests on the assumptions that a restricted subset of English can capture and encode knowledge sufficiently well for a practitioner in the subject area (domain) to read and understand it, while at the same time retaining a sufficiently formal structure for mechanical interpretation by computer. Such a language can greatly facilitate communication between the knowledge engineer who must accurately capture relevant expertise and the domain expert whose expertise determines the final system.

An English-like programming language need not capture all of the subtlety and richness of English. Experts in a field tend to develop a shorthand, or jargon, for commonly held concepts or beliefs, which permits them to communicate with one another rapidly and effectively. Jargon also serves as a rather formal notation for expressing domain concepts, and ROSIE allows a notation close to the normal jargon of experts to be used for this purpose. For example, the following rule shows how the language of lawyers maps onto ROSIE[1]:

[RULE1: RESPONSIBILITY FOR USE OF PRODUCT]
IF the use of the product (at the time of the plaintiff's loss)[2]
is foreseeable,
and that use is reasonable-and-proper
or that use is an emergency
or (there is a description by the defendant of that use
and that description is improper)
or there is no description by the defendant of that use,
[THEN][3]*assert the defendant is responsible for the product's use.*

---

[1]Modeling legal decisionmaking, a major application of ROSIE, is described in Waterman and Peterson (1981).

[2]Parentheses are used in ROSIE rules to eliminate possible surface-level ambiguities, such as determining which description is modified by a prepositional phrase. This particular set of parentheses specifies that *of the plaintiff's loss* modifies *time* and not *use* (as would be the case were the parentheses excluded). The parenthesized clause following the second *or* serves to group the two enclosed sentences as a singular logical unit.

[3]Square brackets surround comments in ROSIE.

The basic ROSIE rule form, *IF conditions, actions*, represents *prescriptive* expertise, which states what to do in various circumstances, rather than merely stating facts. We refer to facts as *descriptive* knowledge.

WHAT ARE RULE-BASED EXPERT SYSTEMS?

ROSIE was designed for building systems that reason symbolically. Many such systems incorporate and act on knowledge or expertise that is normally associated with human experts. The human expert knows facts, or assertions, about his area of expertise (e.g, medicine, geology, the bond market), and he knows rules of inference that allow him to reason within that domain. The rules of inference are not guaranteed to produce the desired answer. They are not formal algorithms, but *heuristics*—rules of thumb or appropriate guides to reasoning. We call systems based on an expert's knowledge of his domain *expert systems*.

Rule-based systems provide an appropriate methodology for implementing expert systems. Rules are a natural formalism for capturing expertise, and they have the flexibility required for incremental development. As a problem changes or the programmer's perception of it changes, a rule-based system can be modified or extended gracefully, whereas traditional structured programs often must undergo drastic restructuring to accommodate new models or situations. For example, if the legislature were to pass a flurry of new laws that affect *the responsibility for the product's use*, we can extend our rule-based legal reasoning system by inserting the new laws as rules and possibly removing old rules or reestablishing precedence among the rules.

Rule-based expert systems[4] contain three main components: (1) a *database* of facts or assertions about some subject matter; (2) a *set of rules* of the form *IF conditions, action* (or *assertion*); and (3) a *monitor* (sometimes called an *inference engine*) that executes a set of rules, given a database. A monitor determines which rules can fire, resolves the conflict if more than one rule can fire, and then executes the chosen rule.

ROSIE supports all three components of a rule-based expert system. Assertions and denials modify a *database* of facts. *Rulesets* group rules

[4]We distinguish *rule-based* expert systems from those based on the predicate calculus or systems of intercommunicating objects. Although a ROSIE user can construct various types of expert systems, to simplify the present discussion we restrict our attention to rule-based systems.

into meaningful chunks. The user labels a ruleset with a particular word, either a noun, a comparative verb, or an imperative verb. Each ruleset also has an associated *monitor*. ROSIE provides three different kinds of monitors for controlling the execution of rules within a ruleset: sequential, cyclical, and random monitors. With planned future developments, ROSIE will allow an expert system developer or knowledge engineer to tailor-make monitors. This freedom will give the knowledge engineer tremendous flexibility. He will be able to create control strategies that meet his exact needs, including monitors that perform forward chaining (deducing consequences from the facts in the database), backward chaining (goal-directed invocation of rulesets that attempt to prove a goal either by further invocation of rulesets or by directly finding the fact in the database), or mixed initiative strategies that combine both forward and backward chaining.

ROSIE PHILOSOPHY AND ARCHITECTURE

ROSIE is a general-purpose tool for writing expert systems. It can support essentially any control strategy and data organization. A well-organized expert system written in ROSIE can represent logic and data in a highly readable form. Such readable code greatly facilitates interaction between the domain expert and the expert system developer.

We chose readability over writability as a major feature of the ROSIE language. The two concepts are by no means equivalent. In fact, as ROSIE moves closer to English, its readability may actually make the language more difficult to write. Negative interference with English (the inability to remember whether a construct is valid ROSIE or valid English) can make writing ROSIE programs difficult. Understanding these tradeoffs is an important aspect of the ROSIE research project.

ROSIE provides rich, expressive power for creating and manipulating a database of concepts. It supports adjectives, prepositional phrases, and relative clauses inside descriptions to generate sets of elements; it supports class structures and concepts of set membership and inclusion; it supports both transitive and intransitive verb phrases; it supports string pattern matching, the definition of rulesets and predicates, and other important capabilities. Recent features include the ability for a database to be shared by multiple ROSIE programs, the definition of demons (programs that awaken to perform some actions when an event occurs, such as searching the database for a specific proposition), and the introduction of meta-structures to provide ROSIE

programs with a self-referential capability (the ability to describe, access, modify, and use linguistic structures).

A particular ROSIE expert system, also referred to as a performance program, consists primarily of the ruleset memory and the active database. The ruleset memory stores the performance program's rules of inference, the *prescriptive* knowledge that provides the system with the intelligence it needs to solve particular problems. The active knowledge base stores the program's *descriptive* knowledge, the facts that specify the particular problem requiring solution and the intermediate conclusions reached by the performance program. Additionally, a performance program may use multiple databases, although only one database can be active at any time.

At a minimum, a performance program uses the ruleset memory and the active database. More elaborate programs also incorporate one or more inactive databases. Multiple databases can serve a variety of purposes. For example, they can represent alternative viewpoints or perspectives concerning a problem. Thus one database might represent the viewpoint of one corporation or superpower, and the other databases might represent other corporations or political regimes. The performance program can switch from database to database, analyze each perspective, and assess the various viewpoints. Alternatively, a knowledge engineer might wish to segment knowledge along a functional, structural, or procedural line. For example, a generalized diagnostic system could use the active database to drive its deductive component. One database might hold general diagnostic knowledge, and other databases might contain symptomatic and diagnostic information relevant to a particular subsystem. With the general database active, the system could determine which system subcomponent failed. The system could then activate the relevant subsystem database and continue the diagnosis in a more constrained environment.

EXAMPLES

Examples provide a good introduction to ROSIE. Our first example is an expert system fragment that plans a camping trip. The expert system's user provides the weather conditions, the length of stay, and other parameters that describe his outing. The expert system uses this information to decide what equipment, food, and clothing the camper should bring along on the trip. The expert system consists of two parts: (1) the invariant camping knowledge captured as facts, and (2) the pragmatics of choosing equipment, food, and clothing, captured in rulesets.

We start with some of the facts. The following assertions build a (partial) database of facts:

Assert t-shirt is a light layer and
flannel shirt is a second layer and
light sweater is a cool third layer and
heavy sweater is a warm third layer and
down-jacket is a coat.

Assert each of the light layer, the second layer, the warm third layer and the coat is a piece of clothing (in a layered top for winter).

Assert 'Layered clothing is effective against the cold' is a winter reason for (any piece of clothing (in a layered top for winter)).

Assert 'Cold nights' is a summer reason for the cool third layer.

This set of assertions establishes some important knowledge concerning camping. Campers know that multiple layers of light or medium-weight clothing provide more protection against the cold than a few pieces of heavy or thick clothing. Campers also know that layered clothing allows them to regulate heat loss by adding or removing clothing. The assertions establish some of this information. They also include justification for some of the equipment. The first assertion defines the concept "layered clothing" and associates a particular piece of clothing with each layer. The second assertion identifies the layer of clothing that a camper should use in wintertime. The third assertion provides the expert system with a justification for using layered clothing in wintertime. The fourth assertion provides a rationale for including a light sweater in the clothing list for summertime.

This set of assertions also illustrates ROSIE's linguistic constructs. The first assertion shows how ROSIE can group related assertions into one rule, using the *and* conjunction. The second assertion introduces the *each of* iterator. An *each of* iterator causes its surrounding action (in this case *assert*) to repeat once for each of its terms—an assert using the light layer, another using the second layer, and so on. The third and fourth assertions show some of the complexity that ROSIE can represent.

The second part of the expert system defines the reasoning used in selecting equipment, food, and clothing for the trip. We capture this knowledge with ROSIE's rulesets. A procedural ruleset illustrates how we define the imperative verb *choose-warm-clothes*:

```
To choose-warm-clothes:
[1] Let the clothing-list be <>.
[2] For each piece of clothing (in a layered top for winter),
      add that piece to 'the clothing-list'.
[3] Assert the winter reason for (any piece of clothing (in a
      layered top for winter) is preferred.
[4] Add (a warm pair of pants) to 'the clothing-list'.
[5] Add (a heavy pair of socks) to 'the clothing-list'.
[6] Add (the hiking footwear) to 'the clothing-list'.
[7] Assert the winter-spring reason for the hiking footwear is preferred.
[8] For each winter accessory,
      add that accessory to 'the clothing-list'.
[9] If the weather will be turning rainy,
      add (the rain-gear) to 'the clothing-list'.
End.
```

A top-level procedure, *produce a checklist*, invokes the *choose-warm-clothing* ruleset as well as other rulesets to create our camping checklist:

```
To produce a checklist:
[1] Gather constraints.
[2] Select the month:
      <'NOV', 'DEC', 'JAN', 'FEB'>
            let the season be winter and choose-warm-clothes;
      <'SEP', 'OCT', 'MAR', 'APR'>
            let the season be fall-spring and choose-moderate-clothes;
      Default:
            let the season be summer and choose-cool-clothes.
[3] Choose-food.
[4] Choose-equipment.
[5] Print-checklist.
End.
```

First, *produce a checklist* calls *gather constraints*, a ruleset to determine the type of camping trip and the conditions surrounding the trip. Then, depending on the time of year and anticipated weather conditions, it chooses appropriate clothing, food, and equipment. Finally, it displays the choices and, if needed, justifies them.

These examples illustrate ROSIE's readability. Later sections of this report describe some of ROSIE's major linguistic structures and provide examples of their use. More substantial examples of systems written in ROSIE may be found in Callero et al. (1984), Fain et al. (1982), and Waterman and Peterson (1981).

HISTORICAL PERSPECTIVE

ROSIE has been an ongoing research effort at Rand since 1979. The language has grown and evolved in many ways over the years. We have worked at improving its expressive power without sacrificing its readability, at regularizing its grammar without sacrificing its expressiveness, and at extending its semantics without introducing new complexities.

The historic precursor to ROSIE was the Rand Intelligent Terminal Agent, RITA (Anderson and Gillogly, 1976; Anderson et al., 1977). Influenced by the success of early rule-oriented styles of knowledge representation and the appeal that their English-like explanation facilities had for users, RITA was a first attempt at making rule-based programming languages easier to use and understand. Production rules in RITA were defined using an English-like syntax with a restricted set of options. RITA's database consisted of object/attribute/value triples. Its monitors allowed either pattern-directed control (forward chaining) or goal-directed control (backward chaining). Although its syntactic and expressive power was limited, RITA showed that a stylized form of English could describe procedural knowledge in rule-based languages.

The preliminary ROSIE design (Waterman et al., 1979) was proposed as a logical extension of RITA. The proposal outlined the deficiencies in RITA and described how they might be overcome. RITA was developed in the C programming language on a PDP 11/45 minicomputer. This limited environment severely restricted RITA's design. To circumvent the problem of RITA's inability to scope rules, ROSIE introduced the concept of *rulesets*.

ROSIE was initially developed using Interlisp on a DECSYSTEM-20. The implementers of early versions of ROSIE adopted several of RITA's best features, such as its input/output (I/O) pattern matcher. They also extended RITA's expressiveness and semantics. By 1981, the ROSIE design was relatively stable, and we began developing in-house applications. We also distributed copies of ROSIE (Version 1.0) to sites outside of Rand.

The first released version of ROSIE included direct support for many special-purpose operations. These were hardwired into ROSIE's grammar, as they are in other programming languages, because they did not fit into ROSIE's general linguistic structure. Some operations required special arguments, and others performed actions that were considered expedient in a programming language.[5] As the number of

[5]These operations included large-grained database operations (*dump, activate, clear, deactivate,* etc.), I/O operations (*open, close, send, read,* etc.), utility operations (*dir, type, delete, copy, parse, build, load,* etc.), and other miscellaneous operations.

special-action verbs began multiplying, the grammar grew increasingly complex, and the need to simplify (or orthogonalize) it became apparent. Examples of orthogonalization are the removal of the distinction between system-defined and user-defined operations, and the introduction of new data elements, i.e, *patterns*, to eliminate the need for special arguments.

The ROSIE system eventually outgrew the capacity of the DECSYSTEM-20, and in 1982, the development moved to VAX-Interlisp on a VAX 11/780 and Interlisp-D on the Xerox SIP 1100 (Dolphin). In 1983, we distributed ROSIE (Version 2.3) and began simplifying the language, expanding its functionality, and improving its performance. Two parallel efforts since that time have been the porting of ROSIE to Portable Standard Lisp (PSL) and the development of a ROSIE compiler in C. We have also examined the needs of ROSIE users and added several new concepts, including *meta-elements*, *shared databases*, and *demons*.

We are currently examining ways of turning ROSIE into its own meta-language and optimizing the resulting code. Future releases of ROSIE will retain the strong features of the current ROSIE, while extending the language's expressiveness and power.

II. THE ROSIE LANGUAGE

We have pursued several different objectives in designing the ROSIE environment. Our primary goal has been to create a language that encourages writing very readable code. To achieve this, we adopted two design criteria: minimality and completeness. These criteria embody two ordinarily competing qualities. Minimality argues against redundancy and verbosity, while completeness requires broad coverage of English-language structures. The net result is a language that serves as a general programming language for a very large range of tasks. It does not, however, make any particular kind of programming task trivial (contrary to what might be expected from a language designed specifically for a narrow class of problem-solving tasks).

We have used natural English as our guide wherever possible. Of course, English has many features that resist translation into precise computational interpretations. Nevertheless, we have given reasonable and relatively natural interpretations to a large number of complex linguistic constructs such as prepositional phrases, relative clauses, and sequences of adjectives. ROSIE recognizes and treats specially many English function words, such as articles, quantifiers, prepositions, and auxiliary verbs. However, ROSIE cannot distinguish content words, such as nouns and adverbs, because it does not understand their meaning. ROSIE has only a superficial knowledge of English, so the user has responsibility for insuring the appropriate application of the surface language to support the desired semantic interpretations.

In our attempt to achieve maximal readability, we have replaced some anachronistic forms of programming diction. Most importantly, we replaced "If *condition* then *action* else *alternate_action*" with "If *condition*, *action*, otherwise *alternate_action*." Those familiar with computer languages, and rule-based programming languages in particular, may initially find this design choice problematic. However, this return to proper English conforms to the principal design heuristic behind ROSIE: Let English be your model.

We divide our discussion of the ROSIE language into two areas. First, we describe ROSIE's fundamental structures—its lowest-level representations for knowledge. Second, we describe ROSIE's linguistic structures—the expressive qualities of the language.

FUNDAMENTAL REPRESENTATIONAL STRUCTURES

A knowledge engineer represents a domain expert's knowledge, using ROSIE's fundamental structures. These structures combine with one another to form more complex structures. Ultimately, they are stored in databases and rulesets. As mentioned earlier, databases store descriptive knowledge, and rulesets store prescriptive or procedural knowledge. ROSIE represents descriptive knowledge as a set of primitive propositions in a database. The propositions might include the initial conditions of the problem, intermediate inferences drawn in the course of the expert system's computation, and final conclusions. ROSIE represents prescriptive knowledge by means of rulesets that consist of a monitor and a set of rules. The prescriptive knowledge describes how to solve a problem, given some facts. Each ROSIE-based expert system consists of one or more databases and one or more rulesets.

ROSIE uses its fundamental structures to store both types of knowledge. Though a ROSIE programmer creates both descriptive and prescriptive structures when programming, the program can manipulate only descriptive structures. ROSIE performance programs cannot yet access, modify, or generate new prescriptive information (i.e., rules).

ROSIE's fundamental structures include *basic elements*, *basic relational forms*, *meta-elements (propositions, intentional descriptions, intentional actions)*, *databases*, *shared databases*, *rulesets*, *demons*, and *monitors*.

Basic Elements

ROSIE's elements define its space of concepts. These elements—ROSIE's datatypes—include *names*, *strings*, *numbers*, *tuples*, *class elements*, and *patterns*.

The *name* element allows ROSIE programs to represent literal names consisting of one or more words, for example, *John*, *Ship #3*, and *Tom Thumb*. Multiword names provide a knowledge engineer with considerable flexibility in naming objects and concepts.

The *string* is a sequence of characters delimited by quotation marks (i.e., double quotes). Strings distinguish between upper- and lower-case characters and allow a greater range of expression than ROSIE's other elements. ROSIE provides operators, such as substring and concatenation, for manipulating strings.

ROSIE supports three types of *number* elements: simple numbers, unit constants, and labeled constants. A simple number is the familiar

datatype found in most programming languages—an integer or floating point number, such as *10* or *2.718*, with no units or labels. Unit constants and labeled constants, on the other hand, are more complex and unique to ROSIE. A unit constant is a number followed by some composite units of measure, which can be combined under multiplication, division, or exponentiation. ROSIE carries the units along in computation and correctly manipulates them, e.g.,

```
Display 88 KM/HR * .625 M/KM.
  55.0 M/HR
Display 9.8 M/KG*SEC↑2 * 5 KG.
  49 M/SEC↑2
```

A labeled constant is a number prefixed by an arbitrary number of tokens, called a label. For example,

```
probability .4
time frame 19
```

are both labeled constants. Units and labels improve the expressiveness and readability of numeric computations. This capacity enhances the representational power of numbers, making their occurrence in ROSIE code meaningful.

The *tuple* combines a list of elements into a single structure. Any member of the ordered tuple can be of any element type including another tuple, for example,

```
<Raoul Wright, "Are you sure?", 2.4 children/family>
<matrix, <2, 3, 1>, <1, 3, 4>, <3, 3, 5>>
```

The *class element* specifies an entire set of elements. It is specified by a ROSIE description indicating which elements belong to the class, for example,

```
any number
any man whose father is ugly
any child where that child does wear sneakers
```

The *pattern* was formerly a special ROSIE construct available only for I/O and matching operations. But in an effort to simplify and clarify the language, we promoted patterns to full element status. They remain the key construct underlying I/O and string manipulation. Additionally, a user may now assert them and thereby put them into the database or pass them as arguments to rulesets.

A pattern is a sequence of subpatterns enclosed in braces and separated by commas. Each subpattern in turn represents a restriction on the successive portions of the text string. The pattern either generates a string or matches against a string. For example, the subpattern *3 blanks* represents a sequence of three blank characters, and the subpattern *one or more numbers* represents a sequence of one or more numeric digits. Some samples of patterns will help illustrate the underlying concepts:[1]

Pattern	Matching String
{*anything, "Fred", anything, end*}	*I know Freddy Smith* *Freddy can't come* *Fred*
{*something, "Tina", anything, "k"*}	*My Tinak* *Tiny Tina can blink*
{*3 or more letters,* {*":"|";"|","*}, *1 blank,* *1 or more numbers, return*}	*file: 1245<carriage-return>* *los angeles, 90025<carriage-return>*

Besides the "standard" types of elements, ROSIE also supports more unusual elements, such as propositions, intentional descriptions, and intentional actions which we call *meta-elements*. Meta-elements extend the kinds of knowledge that can be expressed, represented, and manipulated. Though meta-elements are elements, and thus descriptive in nature, they capture intents rather than fixed quantities. Before we can discuss propositions, the first of the three meta-elements, we must introduce ROSIE's basic relational forms.

Basic Relational Forms

ROSIE relies on five basic relational forms to denote *class membership*, *predication*, *predicate complements*, *transitive verbs*, and *intransitive verbs*. The basic forms and an example of each follow:

element *is a* class-noun
Australia II is a vessel

element *is* adjective
Australia II is seaworthy

element *is* predicate-complement element
Australia II is slightly underpowered

[1]The first pattern uses the keyword *anything*, which matches zero or more characters. The second pattern uses the keyword *something*, which matches one or more characters.

element *does* verb
Australia II does float

element *does* verb element
Martin does sail Australia II

We may also negate each relational form (for example, *Australia II is not seaworthy*), and we may include tense (for example, *Australia II was seaworthy*). Tense information creates a separate and distinct relational form, e.g., *is a man* is independent of *was a man.* The "*is a* class-noun," "*is* adjective," and "*does* verb" relational forms all specify unary relationships. The "*is* predicate-complement element" and "*does* verb element" relational forms specify binary relationships.

We can extend the number of elements in a relationship by appending prepositional phrases, as in

Australia II is moored in Newport for the race
Australia II is rapidly drifting toward shore
Australia II does not have favor with Mr. Connors

The primitive relational forms provide the core for *propositions.*

Propositions

A proposition captures a basic relation as an element. ROSIE delimits a proposition by enclosing it in single quotation marks, for example,

'Martin Scheider did punish Bill Mark in class'
'7 is a prime'

Since ROSIE permits a performance program to access, manipulate, and relate elements, propositions permit a knowledge engineer to operate with basic relations. For example, we can use the proposition *'Warren is interested in T-bills'* as an element in an assertion about a bank's belief system:

Midbank does believe 'Warren is interested in T-bills'

With this assertion in a database, a performance program can now access and manipulate this belief in a variety of ways, for example,

If the bank does believe any thing,
consider that thing as unreliable.
Assert every proposition that Midbank does believe.

The proposition provides a powerful capability to extend the range of problems ROSIE can handle. We demonstrate another aspect of the power that propositions provide by writing an inference engine in ROSIE, which uses propositions to represent rules in a database. Rules 4 and 7, taken from an animal classification system, exemplify this:

[rule 4] Let the consequent of a new rule be 'animal is a bird'
and assert each of 'animal does fly'
and 'animal does lay eggs'
is an antecedent of that rule.

[rule 7] Let the consequent of a new rule be 'animal is an ungulate'
and assert each of 'animal is a mammal'
and 'animal does have hoofs'
is an antecedent of that rule.

The two assertions[2] not only encode the animal classification knowledge, but they also construct a meta-language that we can exploit in writing inference engines. Specifically, they provide a language that permits us to access the antecedent and the consequent of a rule independently.

Our illustrative inference engine provides a mixed-initiative inference capability. In this type of inference engine, we separate knowledge into facts and goals. We have a solution when we can infer the goals from the facts. The inference engine can use the rules in the database to chain forward from facts and backward from goals. It identifies all the rules that can fire in either the forward or backward direction, then discriminates among them to choose the next rule to apply. If it prefers one rule over all the others, it applies that rule; otherwise it stops. The top-level loop of our inference engine is

To infer:
Execute cyclically.
[1] Discriminate among every rule that is capable of firing.
[2] If there is a preferred rule, apply that rule.
End.

The *infer* ruleset uses a number of subsidiary rulesets. One, a predicate named *is capable of firing*, decides if a rule can fire by checking its antecedents against the fact space and the rule's consequents against the goal space. If all the rule's antecedents match the fact space or all

[2]The *let* construct provides a variant method for asserting information into ROSIE's database.

of its consequents match the goal space, the rule can fire. The ruleset's code follows:

To decide a candidaterule is capable of firing:
[1] If every antecedent of the candidaterule is a fact,
assert the candidaterule is chaining forward
and conclude true.
[2] If every consequent of the candidaterule is a hypothesis,
assert the candidaterule is chaining backward
and conclude true.
[3] Conclude false.
End.

After *infer* identifies all the rules that can fire, it selects the rule that is most preferred.[3] The *infer* ruleset presents the candidate rules for firing to *discriminate* one at a time. Thus, *discriminate* must decide only between the current most-preferred rule and the newest rule under consideration. The code for *discriminate* is

To discriminate among a newrule:
[1] If there is a preferred rule,
if the newrule is preferable to that rule,
deny that rule is preferred
and assert the newrule is preferred,
otherwise
assert the newrule is preferred.
End.

Intentional Descriptions

An *intentional description* is an implicit reference to a class of elements. ROSIE descriptions (discussed later) are normally used in conjunction with a determiner or quantifier to immediately access one, some, or every member of a class of elements. Intentional descriptions provide a mechanism by which this access process can be suspended. In a sense, intentional descriptions act as pointers to element sets, serving a function similar to that of call-by-name in ALGOL.

Intentional descriptions take the form of a description prefixed by a determiner and delimited with single quotes, such as,

'the equipment list'
'a command'

[3] In our case, we choose forward chaining over backward chaining and lower-numbered rules over those with larger rule numbers. However, by changing the *is preferable to* predicate, *infer* can use another criterion for choosing a rule.

Intentional descriptions permit knowledge engineers to represent and relate indefinite elements (e.g., *a plan, a target, a counter*) and generic concepts (e.g., *the product's use, the blue side, the clothing list*).[4] The explicit elements referenced by an intentional description may or may not exist (i.e., the set of elements described may be null).

The set elements referenced by an intentional description can be accessed via the phrase *instance of*. Thus, *the instance of 'the clothing list'* might produce the tuple *<PARKA, HAT, T-SHIRT>*, while *every instance of 'a target at any airfield'* might produce *RUNWAY, MUNITIONS SOFT, POL SOFT,* and *MUNITIONS ASSEMBLY AREA*.

The call-by-name facility in ROSIE permits rulesets to affect global relationships specified by an intentional description. As an illustration, consider providing a generic facility to add elements to an existing set. The ROSIE rule

If the weather will be turning rainy,
add (the rain-gear) to 'the clothing list'.

uses the intentional description *'the clothing list'* as an implicit reference to a tuple of elements. The *add* routine

To add an item to a list:
[1] Let the instance of the list be
the concatenation of (the instance of the list)
with <the item>.
End.

then accesses the explicit instance of that tuple and modifies it to include a new item.

Intentional Actions

The last meta-element, the *intentional action*, enables a knowledge engineer to represent an unexecuted action in the database and then perform that action at a later time. Intentional actions provide the raw material necessary to build systems that make plans and then execute those plans. Intentional actions also provide the raw material for building rudimentary simulations.

[4] ROSIE's indefinite article was an early attempt to provide such a facility. An indefinite description (e.g., *a truck*), when used for the first time, would add the relation *TRUCK #1 is a truck* to the database. Thereafter, *the truck* would evaluate to *TRUCK #1*. Although this technique proved adequate for many cases, attempting to represent multiple indefinite trucks led to difficulties. While multiple instances could be asserted in the database, it was not possible to control how the different indefinite relations were used.

An intentional action suspends the invocation of an imperative verb. A possible invocation of the imperative verb "move" might be

Move USS Nimitz from Le Havre to New York.

However, if this statement is used as an intentional action (i.e., is delimited by single quotes, as in *'Move USS Nimitz from Le Havre to New York'*), the intended maneuver is suspended and the knowledge engineer can use it in a relationship. The knowledge engineer could, for example, store it in the database using the following:

Assert 'Move USS Nimitz from Le Havre to New York'
is an action for time 100.

Later the performance program could evaluate the intentional action using the action

Evaluate every action for time 100.

The meta-elements—propositions, intentional descriptions, and intentional actions—represent three of ROSIE's linguistic structures. Given a proposition, a ROSIE program can assert it into the database, remove it from the database, or test for its presence in the database. The intentional description stores an unevaluated description or access operator. That description can represent an indefinite concept or a particular relationship in the database. A ruleset can use intentional descriptions to change or retrieve the elements associated with a particular relationship in the database and to direct other rulesets to the same relationship. The intentional action stores an unevaluated imperative. Access to unevaluated actions permits ROSIE programs to plan, capture intended actions, and then later perform those actions.

Databases

ROSIE uses its database to store facts about the world as well as intermediate computational results. These facts and intermediate results must be *propositions*, which are stored using a three-valued logic system. ROSIE may store a proposition in the database as true, or as false, or the database may not contain that proposition at all (which ROSIE interprets as indeterminate). The three-valued logic provides ROSIE with an "open-world assumption," which implies that ROSIE may not have complete information about a particular situation and will not infer truth or falsity from the absence of a relevant proposition.

The user adds to the database by asserting new propositions or by defining particular values for named elements:

Assert Australia II is a vessel.
Let the vessel be Australia II.
Assert Australia II is not a loser.

ROSIE provides only limited support for handling contradictions. If the user asserts *Australia II is a vessel* and then asserts *Australia II is not a vessel*, a simple contradiction occurs. Only the latter of the two assertions will appear in the database. ROSIE stores propositions as a basic relationship with an attached truth value—a proposition is either provably true, provably false, or it does not exist in the database.

Asserting the negation of a proposition only changes its truth value. The user must deny the proposition to remove it from the database:

Deny Australia II is a vessel.

Conditionals allow users to check a database for the truth or falsity of a proposition:

If Australia II is a vessel, . . .

The database plays a central role in ROSIE. Every assertion results in a database store command, and most conditions require testing the database. By modularizing the database in the design of ROSIE, we have laid the groundwork for independent system improvements. In particular, advances in database technology may ultimately feed into future implementations of ROSIE systems.

Most expert systems written in ROSIE contain more than a single database. The standard database in ROSIE is named GLOBAL. For complex applications, the user may activate other databases and operate on propositions stored within them. Users with very large databases may be able to modularize them and separately dump, restore, and activate each portion as needed. Multiple databases may also function to maintain separate contexts, or worlds, for hypothetical reasoning.

Special system rulesets have been written to simplify testing, adding, and removing propositions to designated databases. The following example illustrates some of these concepts:

Add 'US inflation rate is too high' to European viewpoint.
Add 'European opinion is unnecessarily negative about US economy'
to US viewpoint.
If 'US inflation rate is too high' is true in European viewpoint
and the US's inflation rate > 12 percent,
remove 'European opinion is unnecessarily negative about US economy'
from US viewpoint.

This example refers implicitly to the GLOBAL database and explicitly to the *US viewpoint* and *European viewpoint* databases.

Shared Databases

Distributed expert systems, also called distributed heuristic agents (Sowizral, 1983), consist of multiple expert systems that consult with one another to solve a common problem. ROSIE provides support for building such systems with a mechanism called *shared databases*.[5]

A shared database acts like a normal database: It stores relationships; it allows a program to make assertions, denials, and conditional tests; and it may be active or inactive. But beyond these expected activities, a shared database links together the heuristic agents that share it. Like the blackboard model of HEARSAY-II (Erman and Lesser, 1975), a shared database provides a common, consistent, but changing scratchpad for use by multiple agents. Unlike the blackboard model, agents can concurrently access and modify a shared database, and, also unlike the blackboard model, only those objects "sharing" a particular database have access to it.

ROSIE ensures that each agent sharing a database has exactly the same information as any other agent sharing that database. A change made to a shared database by one agent is visible to all the other sharing agents. ROSIE does not ensure that each agent has identical information at all times, but rather, that each agent's shared database "experiences" the same changes in exactly the same order that the other agents experience. All agents do not see a particular change at the same time, nor is the elapsed time between any two changes at one agent necessarily equal to the elapsed time between the same two changes at some other agent. Nevertheless, all the agents receive the same set of stimuli in the same order and in roughly the same time frame, so it is quite easy for a knowledge engineer to write deadlock-free code.

The shared-database facility hides the many vagaries of distributed programming from the knowledge engineer. He can develop distributed heuristic agents without worrying about concurrency issues such as the arbitration of concurrent updates to the shared database, ensuring reliable communications, and global consistency. The shared-database facility provides knowledge engineers with an effective mechanism for writing distributed heuristic agents, and, when used in conjunction with the demon facility (described in a later section), for constructing fairly intricate control structures with little difficulty.

[5]The shared-database facility exists only for the Xerox SIP 1100 version of ROSIE, not the VAX-Interlisp version.

Description. A shared database has a name that identifies it globally. An agent can turn a database into a shared database by executing

SHARE DATA IN database-name

The database may or may not contain information. If no shared database with that name exists in the system, then the contents of the database become the initial contents for the shared database. If the system already contains a shared database with that name, however, this agent simply joins that community and receives a copy of that shared database. In the process, the agent loses whatever it had in its database at the time it invoked the "share data in" action.

An agent can stop sharing a database by executing

LOCALIZE DATA IN database-name

After this action executes, the specified database no longer shares its contents with other agents in the system. It retains all the information it contained at the time the agent "localized it," and any changes to it will not affect the shared database with the same name; likewise, changes to the shared database will not affect the now-local database.

The remaining shared-knowledge-base operations are identical to the operations permitted on a normal database. A knowledge engineer can assert sentences, deny sentences, and test sentences against the knowledge base. The assertion or denial of a sentence completes immediately; however, because a modification gets sequenced globally, the change caused by it may not appear until some time in the future. Tests against the database return a value immediately.

An agent can have more than one shared database. An agent can share one of its databases with one set of agents, another database with another set of agents, and another with yet another set of agents. In a sense, agents can belong to multiple committees that interact with one another using a semipublic forum. At one extreme, all agents can share one common database. At the other, two agents can interact privately by sharing a database just between themselves.

Architectures for Interacting Heuristic Agents. We have written several systems of distributed heuristic agents using shared knowledge bases. Among those systems were an intelligent secretary, a concurrent search and rescue scenario, and an adaptive route planning system. The shared-database facility proved more than adequate for communicating among agents. Shared databases provide large latitude for decomposing and organizing complex problems into concurrent, cooperating tasks.

Information fusion presents one problem area for which ROSIE's shared-database facility can provide a possible architecture. For example, several data concentrator expert systems could analyze the raw data that come into the system, identify items of interest, and report these findings to the integrator expert system through the shared database. The data concentrator reduces the volume of information to a manageable level. The integrator sees only the interesting information, but from a much broader perspective. The integrator may perform the required computation by itself, or it may also serve as a concentrator for a higher-level integrator.

Committee problem solving provides another area of interest for using multiple heuristic agents and ROSIE's shared-database facility. The database acts as the committee meeting room. The various agents place approaches, comments, and solutions in the shared database. This common knowledge then drives their individual attempts to solve the problem before them.

Yet another use for ROSIE's shared databases is to permit a knowledge engineer to connect existing expert systems into a conglomerate system. For example, we might have several expert systems that solve problems only in their narrow specialty. Instead of combining them into one monolithic expert system, we can coordinate their concurrent execution with a fourth expert system. That new expert system would then interact with the end user, translate the information he provides into a form suitable for use by the expert systems, and translate the requests presented by those expert systems into queries for the end user.

Rulesets

One of the many features that distinguish ROSIE from other rule-based languages is its facility for rule subroutining. A ROSIE user may control the applicability and context of rules by organizing them into logical units called *rulesets*. ROSIE provides the user with three different kinds of rulesets for modularizing his program: *procedure*, *generator*, and *predicate* rulesets. Each serves a different function; each gets invoked differently; each returns differently.

An example of a procedural ruleset is

> *To move a vessel from a source to a destination:*
> *[1] Deny the vessel is docked at the source.*
> *[2] Assert the vessel is docked at the destination.*
> *End.*

This ruleset essentially updates the database when invoked by a statement such as

Move USS Nimitz from Le Havre to Auckland.

Generator rulesets produce either a single value or all the members of some class. References to generators within ROSIE rules cannot be distinguished from references to the database elements. Thus, someone who reads ROSIE code is unaware of what produces a particular element or set of elements. An example generator ruleset is

To generate a vessel:
[1] Produce every moveable object at every port.
[2] Produce every moveable object under sail.
[3] Produce every steamship.
End.

This ruleset would be invoked by a statement such as

Display every vessel.

Predicate rulesets provide a means for determining the truth or falsity of any ROSIE primitive sentence through direct computation. When ROSIE tests a proposition against the database and the result is indeterminate, it invisibly invokes the corresponding predicate ruleset, if such a ruleset exists. The predicate ruleset can conclude true or false, or it can simply return and thus imply an indeterminate value. An example predicate ruleset is

To decide a vessel is seaworthy:
[1] If the vessel does float, conclude true.
[2] If the vessel does leak, conclude false.
End.

This ruleset would be invoked by a statement such as

If Australia II is seaworthy,
move Australia II from Sydney to Newport.

Each ruleset type corresponds to a particular word class: generators correspond to nouns, procedures to imperative verbs, and predicates to comparative verbs. Rulesets allow domain words to be defined operationally in whatever fashion the knowledge engineer chooses, and only as precisely as necessary.

ROSIE users may additionally define system rulesets, which permit programmers to include Interlisp code in their ROSIE system. System rulesets may not call ROSIE rulesets. Thus they serve mainly to provide access to system parameters such as time-of-day, date, or other important information that ROSIE does not provide directly.

Demons

The word "demon" has come to have a specialized meaning in programming: It refers to a program (or ruleset) that lies dormant until a particular condition occurs, then is activated and takes some action. The ROSIE system samples changes to the database at key stages in the execution of a program. When a test or change occurs that meets a demon's condition for awakening, that demon becomes active. Demons can be used, for example, for tracing and debugging during program development, and for checking database consistency as the database undergoes changes.

The demon facility in ROSIE allows a knowledge engineer to selectively capture control during the course of a performance program's execution. Demons cannot take control of a computation haphazardly; they get invoked at precise points. In ROSIE, they are invoked (1) before a proposition is asserted, denied, or tested against the database; (2) before the database starts generating any elements from a description; (3) before each element gets produced by a generator matching a description; and (4) before invoking an imperative verb. These points in the ROSIE operating cycle define four classes of demons.

A demon can decide whether or not the operation it preempted should occur. It can exit by using a *return* statement and thus prevent the completion of an operation, or it can exit by using *continue*, which causes the preempted operation to complete.

Demons do not capture all assertions, denials, etc. Rather, they capture all assertions, denials, or operations that match their defining forms. Thus a knowledge engineer may selectively intercept the assertion of an *is a man* relation or the testing of an *a person1 does love a person2* relation. A knowledge engineer can exercise precise control over the relations in the database and thus ensure its consistency. For example, the following demon checks the age of any person we wish to define as a man:

Before asserting a person is a man:
[1] If the person's age is greater than or equal to 21,
continue.
End.

If the person is 21 years of age or older, it permits the redefinition; otherwise it does not. A demon must explicitly *continue* for the preempted operation to complete; otherwise, the demon returns and impedes the interrupted operation.

Using Demons for Communication Between Distributed Agents. Demons simplify the communication control structure of distributed heuristic agents. Without demons, a distributed agent must continually poll the shared database to discover any new information. With demons, an agent can respond to changes in the database as they happen.

Distributed agents use shared databases for communicating among one another. Agents *send* new information by asserting or denying propositions in the shared database. They *receive* information by testing for propositions or by generating elements from the shared knowledge base. To write an effective agent without using demons, a knowledge engineer must know what information to look for and when to look for it. An agent's standard structure usually consists of an infinite loop that checks for the presence of the anticipated information in the shared database and, when the agent discovers one of the possibilities, performs the appropriate actions—a process similar to hardware polling.

Demons significantly alter an agent's communication control structure. No longer must an agent check its shared database for new information. The knowledge engineer can write a *before asserting* or *before denying* demon for each of the possible communications. Then, when the information of interest appears in the shared database, the demon processes it directly. This changes an agent's computational structure from a "polling-driven" computation to an "interrupt-driven" computation.

The demon facility permits agents to capture attempts to assert, deny, or test propositions; to start generating elements; to produce elements; and to invoke verbs. However, in a shared database, only global asserts and denies will cause demons to execute. The other operations (testing propositions, beginning to generate elements, and producing elements) work only with an agent's local copy of the shared database. Thus, the demons that do not capture modifications execute only for the agent performing such operations.

A shared database can provide the necessary medium for communication without ever storing any information. This activity could result in a cluttered database, but demons can act as "sentries" that guard the database's contents. By "returning" rather than "continuing," a demon can capture the communication, perform the required computation, and prevent the modification from entering the local database.

Using Demons to Emulate Frames. The demon facility allows ROSIE programmers to write frame-based performance programs. A frame is an abstract specification for a class of entities, which allows programmers to write systems from a behavioral perspective. Each entity, or instance of a frame, consists of a name, attributes, and the associated values of those attributes. Each attribute may also have three routines associated with it. The routines are invoked as a side effect of performing some operation on an attribute's value. The *if-added* routine is invoked whenever a value is added to the attribute; the *if-removed* routine is invoked whenever a value is removed from an attribute; and the *if-needed* routine is invoked whenever an attribute is queried for a value. In this paradigm, computations occur as side effects initiated by the manipulation of an attribute.

We can use the demon facility to manage frames. For each new frame type we can write two demons, one to capture control at the time of the frame's creation and the other to capture control at its destruction. The two demons also create and destroy the frame's associated attributes. For each manipulation of a frame, we must write a demon to capture the intended action and correctly update the affected frames.

Managing a meeting calendar provides a good example of the use of frames. Two different kinds of frames are needed, one to represent a meeting and another to represent a person. Attributes of a meeting include its participants, its starting time, its anticipated duration, its location, and its topic. To use the demon facility in defining a meeting frame with these attributes, we could write

```
Before asserting a meeting is a meeting:
[1] Assert <> is a participants of the meeting
    and assert NOTHING is a topic of the meeting
    and assert <> is a start of the meeting
    and assert <> is a duration of the meeting
    and assert <> is a location of the meeting
[2] Continue.
End.
```

(The corresponding *denying a meeting is a meeting* demon would be defined similarly.)

We can now create meetings with ROSIE's *create* verb, for example,

```
Create a meeting
    and let "Setting priorities" be the topic of that meeting
    and let 8am be the start of that meeting
    and let 1 hour be the duration of that meeting
    and let room 247 be the location of that meeting.
```

When this action executes, *create* first generates a new name element from the description's class noun (*meeting* in this case) and a number. For example, assume that the new name element is *MEETING #34*. After generating a new name element, *create* automatically asserts that *MEETING #34 is a meeting*. The assertion causes the *before asserting a meeting is a meeting* demon to execute. It then asserts *<> is a participants of MEETING #34, NOTHING is a topic of MEETING #34, <> is a start of MEETING #34*, etc. Finally, the demon continues by allowing *MEETING #34 is a meeting* to be added to the database. Now our original create statement continues executing with *let "Setting priorities" be the topic of that meeting*. Because of ROSIE's ability to handle anaphora, *that meeting* evaluates to *MEETING #34*; and because of the demon, *the topic of MEETING #34* already exists, permitting the use of the definite description.

The "person" frame operates similarly. It consists of two fields, the person's name and the person's meetings. The demon is defined by

```
Before asserting a person is a person:
[1] Assert no name is a name of the person
    and assert <> is a meetings of the person.
[2] Continue.
End.
```

With a frame for meeting and another for person, we can now write routines for manipulating these frames. The two activities we wish to illustrate are adding a person to a meeting and removing a person from a meeting. We define a demon that captures the assertion that a person will attend a meeting:

```
Before asserting a name will attend a meeting:
Private subject.
[1] If the name is a name of any person,
        let the subject be that person,
    otherwise, create a person
        and let that person's name be the name
        and let the subject be that person.
[2] Add the meeting to 'the subject's meetings'.
[3] Add the subject to 'the meeting's participants'.
End.
```

This demon first uses the person's name to locate that person's frame. It then updates that person's list of meetings and updates the meeting's list of participants. Rather than continuing and letting a

fact such as *JOHN will attend MEETING #34* enter the database, the demon returns, ensuring that unnecessary information does not clutter up the database. Similarly, we can write a demon that captures the assertion that a person will not attend a meeting:

```
Before asserting a name will not attend a meeting:
Private subject.
[1] If the name is a name of any person,
        let the subject be that person,
    otherwise,
        send {"I know of no person named", the name, ".", CR}
        and return.
[2] Remove the meeting from 'the subject's meetings'.
[3] Remove the subject from 'the meeting's participants'.
End.
```

Monitors

ROSIE associates a *monitor* or control program with each ruleset which specifies the execution order for all the rules in the ruleset. Currently, ROSIE supports three different monitors called *sequential*, *cyclic*, and *random*. These monitors execute a ruleset's rules in standard ways. The sequential monitor executes each rule in sequence and, after executing the last rule, causes the ruleset to return. The cyclic monitor also executes the rules in the ruleset in sequence; however, rather than returning when it executes the last rule, it reexecutes the first rule, and so on. The random monitor repeatedly executes the rules in the ruleset by choosing the next rule to execute at random. Future versions of ROSIE will permit knowledge engineers to write their own monitors.

LINGUISTIC STRUCTURES

Our main goal in developing ROSIE is to provide an understandable programming language. We have used English to guide this development, because it is the language of choice for most domain experts. Ideally, ROSIE should mimic English exactly. However, this extreme position presents two problems. First, an expressive and broadly based language, such as English, allows users to express concepts that might not map directly onto fundamental representational structures (of ROSIE or any other programming language). Second, users often need programming idioms (e.g., variables, *for* loops) that are awkward to express in English.

ROSIE allows programmers to specify complex sentences, but it cannot store such sentences directly in the database. Instead, it decomposes a complex sentence into its fundamental structures. ROSIE's complex linguistic forms extend the limited expressive capabilities of its basic representational forms and give the language an appearance much like that of English.

It is difficult to describe ROSIE's linguistic structure because the language is self-recursive. That is, some linguistic constructs rely on other linguistic constructs that rely on the former constructs. In the following discussion, we first present *terms*, which resemble noun phrases. Terms refer to specific things (i.e., ROSIE elements). A term may be either an explicit element (e.g., *Ronald Reagan*) or a description which refers to an element (e.g., *The United States President*). ROSIE's *descriptions* present a difficulty with recursion, because a description can be modified by prepositional phrases (embedded terms) or relative clauses (embedded verb phrases). Next we consider ROSIE's *verb phrases*, both relational and comparative forms. This leads into a discussion of *sentences* and *conditions*. Finally, we discuss ROSIE's higher-level linguistic constructs, *rules* and *actions*.

Terms

ROSIE *terms* act as noun phrases. Terms permit ROSIE programmers to access, manipulate, and store elements. A term always generates one or more values, which are always elements. When a term evaluates, it becomes the element it generates. For example, when the term *the mayor of Los Angeles* evaluates, it becomes the element *TOM BRADLEY*. Thus, *Assert the mayor of Los Angeles is happy* would assert the proposition *'TOM BRADLEY is happy'* in the database.

Four kinds of terms exist: (1) *element terms*, which consist of the elements and element constructors; (2) *expressions*, which allow a user to compute a numeric quantity; (3) *description-based terms*, which compute values by searching the database and optionally invoking rulesets; and (4) linguistic forms that refer to elements. This last class of terms includes possessive and anaphoric forms.

The element term includes all the elements and the special linguistic forms that create new elements. The number *probability .7* is not only an element but also a number term. Similarly, names, strings, tuples, patterns, propositions, class elements, intentional descriptions, and intentional actions also serve both as elements and as terms. The terms that create new elements look much like actual elements; however, they permit the inclusion of embedded terms. The proposition constructor illustrates this point well: When ROSIE encounters a

proposition term, it evaluates all embedded terms until only elements remain (since proposition elements can contain only a single primitive relational form and elements). For example, the proposition terms

'John Smith was late for work'
'The teacher did punish the student in class'
'3 + 4 is a prime'

could evaluate to

'JOHN SMITH was late for WORK' [6]
'MARTHA did punish JAMES in CLASS'
'7 is a prime'

Expressions also serve as terms. Expressions include the standard arithmetic infix operators as well as unary negation. ROSIE, with its built-in rulesets, also provides transcendental functions and other unary arithmetic operators. Typical expressions terms might include *3 + 4* or *the liquid's volume * the liquid's density*.

Every description-based term consists of a quantifier (*some, every*) or a determiner (*a, an, the*), followed by a description. For example, *every big burly man that does eat quiche* uses the quantifier *every* and implicitly iterates the enclosing action once for each element it generates. Thus, had the programmer used this term in the action *Display every big burly man that does eat quiche*, when ROSIE evaluated the statement, it would execute it once for each element that satisfies the description *big burly man that does eat quiche*.

Descriptions

ROSIE would be quite stilted if it permitted only simple linguistic forms such as elements. A richer, more English-like flavor results from the use of *descriptions*, which represent elements much as variables represent values. A description consists of any number of adjectives followed by a class-noun followed by any number of prepositional phrases and possibly some relative clauses:[7]

The *sleek red vessel*
Every *vessel that does start on time*

When descriptions and the basic relational forms are mixed, the resulting sentences are capable of expressing complex concepts that are both readable and understandable, e.g.,

[6] When ROSIE evaluates a term, it generates an element. ROSIE prints all elements in upper case.

[7] The examples include determiners and quantifiers to make reading the descriptions easier.

The *vessel that does not start on time*
    is not likely to win
Every *eligible vessel that is moored in Newport*
    is likely to race

Descriptions identify a base class (e.g., *vessel*) along with restrictions that narrow that class to the concept of interest (e.g., *seaworthy vessel that is docked in Newport*). In ROSIE, a description is always preceded by either a quantifier or a determiner. This is followed by the description itself, consisting of any number of adjectives, the base class noun with its prepositional attachments, and any number of relative clauses. Thus, a description designates a set of elements generated by the base class noun and constrained by the modifiers.

Since descriptions are used extensively to retrieve and add knowledge, it is important to understand how ROSIE interprets them. ROSIE interprets each modifier of the base class noun independently. The ROSIE assertion

Assert John is a big burly man who does eat quiche.

causes the addition of four propositions to the database:

JOHN is a man.
JOHN is big.
JOHN is burly.
JOHN does eat QUICHE.

ROSIE's interpretations of descriptions can lead to potential pitfalls. For example, with the additional assertion

Assert John is a small fry.

the database would contain

JOHN is a man.
JOHN is a fry.
JOHN is big.
JOHN is small.
JOHN is burly.
JOHN does eat QUICHE.

This database would generate the same element, *JOHN*, when asked *Display every small man* and *Display every big man.*

We also use descriptions to retrieve information from the database. For example, the two actions *Display every seaworthy vessel* and *Display every vessel that is seaworthy* both generate the same elements. Each action first generates the set of all elements that are *is a vessel.* Next, ROSIE prunes this set by checking that each element also *is seaworthy.*

Thus, adjectives and class nouns are closely related but have different semantics. The "*is a* class-noun" relation establishes membership in a set, as in

Fudge is a sweet

The "*is* adjective" form modifies or predicates the element, as in

Fudge is sweet

In the first phrase, we establish *fudge* as a *sweet.* In the latter phrase, we establish that *sweet* describes or modifies *fudge.* ROSIE treats these two uses of sweet entirely separately. Asserting either of these propositions does not affect the other; and ROSIE cannot infer one from the other automatically without the user first establishing some additional relationships (e.g., that any sweet is sweet).

Prepositional phrases in English add specifics to a description. They play a significant role in ROSIE, becoming an integral part of the database relation by specializing the class definition. For example, the three phrases

a spot in the sun
a spot on the sun
a spot

have distinct meanings. ROSIE differentiates among descriptions with dissimilar prepositional phrases even when the class name is the same in all of them.

Relative clauses also add specifics to an English description, and they perform the same function in ROSIE. But unlike prepositional phrases, ROSIE's relative clauses restrict the set of elements generated by a description rather than becoming part of the database relation. ROSIE provides a number of distinct relative clause forms. The following are examples of descriptions with relative clauses and the relationships they represent.

Description:

A *company which is bankrupt*

An *employee of ITT who does play tennis and who did retire*

Relationships represented:

<element> is a company
<element> is bankrupt

<element> is an employee of ITT
<element> does play tennis
<element> did retire

Verb Phrases

Verb phrases provide the mechanism for constructing ROSIE sentences. Together with terms, ROSIE's verb phrases can perform a variety of functions. They can, for example, form basic relationships, compare elements, test propositions against the database, and determine the cardinality of a set of elements.

ROSIE permits five basic verb phrases, each capturing a specific class of English usage, and each mapping onto one of the five relational forms. The first basic verb phrase is *class membership*, using the *is*, *was*, and *will* verbs in conjunction with the indefinite article *a* or *an*. These verb phrases can include *be* and the negative *not*. Examples are

John Smythe *is a doctor*
Bill Walsh *will not be a witness*
Martha Jones *was an individual with glasses*

The second basic verb phrase is *predication*. This is similar to the class membership form, but it replaces the indefinite article with a "relation-name" and can also add "prepositional elements." Examples are

Tom *is happy*
Rena *was not alone at the time*
Carol *will be late (at the sound of the bell)*

The *predicate complement*, the third kind of verb phrase, creates binary relations between the subject of the sentence and the term following the predicate complement. Examples are

Martin *is nuclear powered*
The play *was not really exciting*
US Steel *will be running rapidly toward Bethlehem*

The fourth type of basic verb phrase is the *intransitive verb*, as in

Spot *did eat*
Raoul *does eat with a fork*
Billy *will not eat without a fuss*

The final verb phrase type is the *transitive verb*, as in

Every student *does study biology at school*
Susan *will cook a steak for dinner*
Jill *did not divide by 2*

In addition to the basic verb forms, a few built-in *comparative verb forms* allow comparison of numbers and other elements. Element equality is tested using the equality sentence form, which can be written tersely using the = character or in expanded natural English as *is equal to*:

<term1> is equal to <term2>
<term1> = <term2>

<term1> is not equal to <term2>
<term1> ≃ <term2>

When number elements are tested for equality, they are equal only if both have the same units or labels and represent the same numeric value. Numbers that have different units or labels are not equal regardless of their numeric values. For example, these sentences will all test true:

33.2 = 33.2
40 miles/hour = 40 miles/hour
82 miles = 82.0 miles
17 apples ≃ 17
17 apples ≃ 17 oranges

ROSIE includes relational forms for comparing number elements, e.g., *greater than*, *not greater than*, *greater than or equal to*, *not greater than or equal to*, and similarly for the *less than* comparisons. Comparisons can be made only between numbers with the same units or labels. All other comparisons are illegal and will generate errors. Comparative verb forms are *not* basic verb forms, and therefore they cannot be used in propositions or asserted into the database.

The remaining legal verb forms make a variety of tests which may or may not involve the database. Some are supplied for convenience;

others expand ROSIE's capabilities in fundamental ways. Like verb forms, these forms are *not* basic sentences and therefore cannot be used in propositions or asserted into the database. They include sentences such as

'John is a student' *is provably true*
'John is not a student' *is provably false*
'John is not exemplary' *is provably true*
'John is exemplary' *is provably false*

These verb forms allow ROSIE to test proposition elements against the database. A proposition is *provably true* if it is found in the database, and it is *provably false* if its negation is found. If neither the positive nor the negative of the proposition exists in the database, the test *is not provably true* and the test *is not provably false* will both test true.

Verb forms that test the cardinality of a class of elements take the form

If *there is a* file for the employee, print that file.
If *there is no* file for the employee, request data.
If *there is just one* enemy ship, attack that ship.
If *there is more than one* enemy ship, surrender.

Sentences

The largest syntactical unit a programmer can add to a database is the *primitive sentence*. However, ROSIE permits the expression of more complex sentences. To add these to a database, ROSIE first decomposes the complex sentence into a set of primitive sentences. As we shall see, conditions, actions, rules, and rulesets all incorporate sentences to effect their results. The constructs discussed above (e.g., terms, descriptions, verb phrases, etc.) are the building blocks of ROSIE sentences.

ROSIE's primitive sentences are those that define a single relationship. They are constructed using the legal relational forms, with terms in place of elements. The following are examples of primitive sentences:

John is a man
The student did not fail the exam
John does support the Republican candidate
Every boy does like some girl
John's father will not succeed in business
Any friendly ship was attacked before 1300 hours

There are also sentences that test the element generated by the term against the constraints of the description. All relationships must test true for the test to succeed. The *not* option negates the result of the test. If the tested sentence is a primitive sentence, it tests true only when a sentence in the database matches it exactly or when a predicate matching the sentence concludes true. For example,

If John is an exemplary student of math, display John.
If John is not an exemplary student who did pass the final and who did pass the midterm, disqualify John.

Other typical sentences are formed with the verb phrases described previously.

Conditions

Conditions are sentences.that occur within the context of a test such as *if*, *while*, or *until*. Boolean combinations of sentences can be formed with the words *and* and *or*. The negation of a sentence is formed by inserting the word *not* in its appropriate location within the verb phrase of the sentence. The Boolean connectives *and* and *or* are given ordinary precedence during parsing, so that *and* groups conditions with higher precedence. To aid readability when several conditions participate in some test, the conditions can be separated by commas or grouped within parentheses.

Two example conditionals are

While John is happy and Mary is sad, . . .
If the value > 0 or the sum is equal to 5, . . .

Rules and Actions

Rules constitute the principal syntactic category of ROSIE. A ROSIE rule begins with the keyword *If* followed by a condition and the rule's associated actions. ROSIE permits degenerate rules that consist only of actions. Degenerate rules are equivalent to *If "true", actions.*

Actions are ROSIE's workhorses. They act upon ROSIE's database; they interact with the user; and they control the system's inferencing. There are several important types of actions, including

1. Actions that determine which propositions are affirmed in the database.
2. Actions that define conditional behaviors.

3. Actions that control iterations.
4. Actions for input and output.
5. Actions to invoke and terminate rulesets.
6. Actions for file management.

We describe each of these in turn.

1. The database contains affirmed propositions, which can be positive or negative, depending on the verb form used. Because ROSIE supports three possible truth-values (true, false, and indeterminate), the user must employ a variety of actions to manipulate the different types of data that can arise. The primary way in which the user affects the database is by asserting sentences. The *assert* action takes as its argument one or more sentences separated by *and*. Each of these is interpreted as one or more primitive sentences, and each is affirmed. When a primitive sentence is affirmed, its negation is denied. The effect of the deny action is to delete the primitive sentence from the database if it exists.

2. ROSIE supports several actions that define conditional behaviors and conditional looping. These include the *if*, *unless*, *while*, and *until* constructs.

3. ROSIE supports a variety of types of iterative actions. The most significant of these is the *for each* action. This action takes a quantified description and an action as its arguments. ROSIE performs the action for the corresponding set of database elements that match the description. The *for each* action can be augmented with optional *until* and *while* conditions which restrict the iteration of the *for each* in the expected ways. The *for each* iteration terminates as soon as the *until* condition is satisfied or the *while* condition is no longer true. The user may also elide the *for each* and retain the *while* or *until* iterator.

4. ROSIE supports very flexible communications with other systems. The basic output action is *send*, and the basic input action is *read*. Send and read can address files, that is, data structures on the local machine. The key construct underlying the I/O and string-manipulation facilities in ROSIE is the *pattern*. A pattern is a description of a string, which can be as simple as a string constant or as complex as a regular expression defining the strings of interest.

5. Programs implicitly invoke predicate and generator rulesets. Each ruleset type exits using distinct actions. Procedural rulesets exit using the *return* action. Generator rulesets exit using either the *produce* or *return* actions. Predicate rulesets exit using either the *conclude* or *return* actions.

6. An entire package of other actions has been provided for interacting with files. The parse action reads a file of ROSIE source code,

checks it for syntactic errors, produces an executable parsed version of the original file, and also produces a text version of the original file that may be edited. When the user *loads* the parsed version of the file, ROSIE immediately executes any rules in the file and indicates which rulesets have been defined. It also establishes the conditions necessary for interactive editing of rulesets.

ROSIE also provides many facilities to edit, maintain, and manipulate files, and it supports other constructs, including flow control actions. These are covered in detail in *The ROSIE Language Reference Manual* (Fain et al., 1981).

III. INITIAL EVALUATION OF ROSIE

MAJOR ROSIE APPLICATIONS

The ROSIE environment has been used for major system development work by Rand projects and by external groups. Three examples illustrate the diversity of these applications: the development of a model of legal decisionmaking (Waterman and Peterson, 1981); the design of the TATR tactical air target recommender (Callero et al., 1984); and the design of an experimental workstation (Adept) to aid combat intelligence analysts and combat operations decisionmakers (Beebe et al., 1984).

The legal decisionmaking expert system demonstrated the appropriateness of using rule-based techniques to encode formal rules of law together with informal rules of procedure and strategy. The project explored a number of techniques for encoding legal knowledge. The final system took a restricted product-liability situation, examined it, and calculated a final dollar figure that represented the amount of money a plaintiff could recover in court.

The TATR system was designed to aid Air Force tactical air targeteers in planning strikes against enemy airfields. The system makes two major interacting choices: It chooses which airfields to strike and which targets to strike on those airfields. The planning system examines its options, evaluates the options against a set of metrics, and chooses the set that maximizes the effect on the enemy.

The Adept Workstation, developed by TRW Defense Systems, is an ambitious effort to aid military analysts in assessing enemy activities, using near-real-time intelligence analysis. Much of the processing of diverse intelligence information involves the use of expert "heuristic" knowledge. The objectives of the project were to verify the functional design of the workstation for situation assessment and to demonstrate the feasibility of applying AI techniques to this domain. The project included an explicit working demonstration system consisting of about 200 rules programmed in a combination of ROSIE and Lisp.

ADVANTAGES OF ROSIE

ROSIE has been in use for major system development for approximately four years. Sufficient experience in its use has been gathered to allow an initial assessment of its advantages and disadvantages.

We believe major advantages have been demonstrated in all of ROSIE's applications. For example, results of an explicit evaluation phase for the Adept Workstation are summarized by Beebe et al. (1984) as follows:[1]

> . . . the capabilities and potential of the AI software implementation was most appreciated. "Most encouraging was the thought patterns in the . . . software." . . . it has "the greatest potential for modification and enhancement." . . . The analysts strongly approved having the code, that is the ROSIE rule base, in English-like form. This "is important if the analyst is to trust the automated system."

The primary advantages of ROSIE are its readability, flexibility, and expressiveness.

Readability. A knowledge engineer can write an application in ROSIE so that a specialist in that application area can read it and understand it. However, a knowledge engineer can also write obscure code in ROSIE—just as programmers can in other programming languages and writers can in natural languages. Whether a reader can understand the final system depends heavily on the overall structure of the program and its knowledge base. If the expert system's architecture matches the problem, the program almost always allows graceful expansion of the system. Often, a domain specialist can make straightforward additions or modifications directly, using existing ROSIE code as a template and modifying the language within that template. On the other hand, if a knowledge engineer writes the expert system with little thought to an appropriate underlying structure, the richness of ROSIE permits writing a diversity of structures that even the most psychic readers would find difficult to understand, modify, or expand.

Readability remains an important concern. Any tool or methodology that improves communication between a computer-trained knowledge engineer and the domain expert with whom that knowledge engineer must interact eases the construction of the expert system. Problems of the interface between the knowledge engineer and the domain expert present a very difficult barrier. If the domain expert cannot verify or understand the translation of his expertise, communication problems are exacerbated. It is clearly advantageous to have a medium that can serve as a common meeting ground for the knowledge engineer and the domain expert to use in discussing ideas and their representation.

Flexibility. Complex models are usually written as traditional computer programs, using deeply nested logic and specially tailored data structures. These models tend to be difficult to change,

[1]The quotations within this excerpt are taken from questionnaires filled out by analysts evaluating the Adept Workstation.

particularly when changes affect the fundamental structure of the program or data. In contrast, ROSIE—like other rule-based languages tailored for the production of expert systems—allows, even encourages, the development of highly modular programs that retain considerable flexibility even as they expand.

ROSIE programs are primarily constructed from a set of conditional rules. ROSIE encourages structuring these programs with named rulesets that cluster rules into operations, generators, or predicates, which chunk important prescriptive knowledge into single units. However, ROSIE programs are predominantly "flat" structures composed of rules and assertions of facts. As such, they permit the graceful introduction of additional rules (or whole rulesets) and assertions. The resulting flexibility in the program's structure permits additions or even "repackaging" of rules into new rulesets that can dramatically modify program behavior without major reprogramming.

Expressiveness. ROSIE is a rich, complex language. It includes a wide variety of linguistic forms and permits the expression of complex concepts. We believe the English-like expressive power of ROSIE goes sufficiently beyond that of traditional programming languages to permit new modes of expression that could conceivably make the difference between the ability to express a model and either not making the attempt or floundering in the process.

DISADVANTAGES OF ROSIE

In its present form, ROSIE is not without disadvantages. We believe there are two primary areas where improvement is needed:

1. *Efficiency.* ROSIE as currently implemented in Interlisp executes too slowly to permit effective development and testing of complex models.
2. *Writability.* It is difficult to rewrite some English-language concepts in ROSIE.

Limited Efficiency. ROSIE was developed without concern for size or speed. Our objective was to ease the development of complex models. Yet models developed with ROSIE quickly grow to such a size that development, debugging, and testing involve frustrating delays. Some of the inefficiency is due to the implementation of ROSIE within Interlisp, which is already a complex system; some is due to the inherent complexity of ROSIE; and some is due to inefficiencies in implementation.

Currently, we are rewriting ROSIE from Interlisp into PSL (Portable Standard Lisp). This will not only circumvent Interlisp's size and complexity, it will also increase the availability of the ROSIE language, since PSL runs on a wide variety of machines. We are also improving the implementation and searching for ways to improve overall execution speed.

Limited Writability. Although knowledge engineers can express many and varied concepts in ROSIE, often quite naturally, they may encounter difficulty in finding appropriate linguistic forms for some concepts. ROSIE's expressiveness and its resulting complexity cause this difficulty. Its proximity to English causes negative interference—a knowledge engineer may try to express a concept using correct English phrasing, only to find that ROSIE does not support that phrasing. Mapping a correct English phrase into correct ROSIE structure may be difficult.

A language is "writable" if a user can learn it easily and can then easily write code that is relatively error-free, both syntactically and semantically. Restricting a language's syntax to a small, orthogonal, and intuitive set of constructs can go a long way toward making a language writable. Such restrictions tend to make a language's rules of composition easier to remember and apply. We are in the process of orthogonalizing the language and are finding minimal but expressive constructs that have already helped make ROSIE more writable.

FUTURE DIRECTIONS

We continue to improve ROSIE by evaluating its strengths and weaknesses, then modifying, extending, or enhancing it as appropriate. User feedback plays an important role in this process. ROSIE has a range of users, each of whom wishes to solve problems differently, to use ROSIE differently. These users are generally either knowledge engineers interested in solving some problem in a particular domain or tool builders interested in providing generic help for performance-program developers. These two user types inevitably merge, to some degree. Knowledge engineers realize that some portion of their system can help other users, and tool builders realize that the act of developing systems provides the perspective necessary to build useful tools. Both types eventually find their implementation language inadequate; most of them want ROSIE to work slightly differently, to provide more control over inferencing, or to provide more expressive representation. Each user has his or her own set of desired "improvements" to ROSIE.

Future changes to ROSIE fall into four categories: (1) expressiveness; (2) support for specialized structures; (3) meta-structures, meta-linguistics, and meta-control; and (4) knowledge-based optimization.

Expressiveness. ROSIE's expressive power is quite broad, permitting the direct representation of a wide range of syntactic constructs. However, it does not cover all of English. Some linguistic forms that ROSIE cannot support, such as passive sentential forms, can be restated in other ways that are acceptable to ROSIE. Other linguistic forms, primarily modals expressing what *could* be the case, or what *should* happen, are presently quite difficult to map into ROSIE. It is very difficult to express thoughts like *John should sue Mary* or *Susan might like windsurfing*. Extending ROSIE's linguistic constructs remains a large part of our research effort and will require careful study to identify graceful ways of extending the language.

Support for Specialized Structures. The primary goal of the ROSIE language research project is the development of a computer language that is directly readable by experts in a discipline. These experts not only use specialized jargon, they also structure their information in specialized ways. For example, information might be succinctly and readably presented in tabular form, such as a decision table (Shapiro et al., 1985). Also, experts often describe situations in terms of a set of objects, each having certain attributes with associated values or attached procedures (McArthur et al., 1984). Decision tables and objects seem sufficiently important to warrant their direct inclusion in ROSIE.

Meta-structures, Meta-linguistics, and Meta-control. Meta-linguistics will enable us to substantially improve the expressiveness of ROSIE without increasing ambiguity. Meta-linguistic constructs permit self-reference, that is, they allow a language to refer to itself. Phrases such as *the antecedent in the third rule* or *the description that references the theory of strict liability* can permit a knowledge engineer to write code that examines and manipulates procedural knowledge. Meta-linguistics and the supporting meta-structures will provide the support necessary to construct programs that reason self-reflexively, that can explain their actions, that can modify themselves, and that can learn.

Meta-linguistics and meta-structures will permit ROSIE users to tailor the language themselves. A full meta-language will allow users to write new monitors and thus control inferencing on a per ruleset basis. For example, the sequential monitor might be rewritten as

To execute a set-of-rules in sequential order:
[1] loop: if the set-of-rules is empty, return.
[2] evaluate the first rule in the set-of-rules.
[3] let the set-of-rules be the tail of the set-of-rules.
[4] goto loop.
End.

Knowledge-Based Optimization. As ROSIE has evolved into a stable and mature programming environment, performance has become a critical issue. ROSIE is too slow for many applications, and as we expand it to include meta-linguistic capabilities, it will slow down even more. To correct this weakness, we are developing methods not only to optimize ROSIE's internal code but also to optimize the code that ROSIE generates when it compiles a ROSIE program into Lisp. We are applying some of these techniques in our rewriting of ROSIE into PSL. One such technique, symbolic evaluation, allows us to expand function calls that have at least one constant argument into an equivalent call on a new function. The new function does not need the constant argument, since it assumes that value. This process of symbolic evaluation generates code that executes more rapidly but may, at times, require more memory. With judicious application of the technology, however, we can increase the running speed of many routines without suffering any cost in memory.

Symbolic evaluation alone cannot hope to achieve the necessary speed improvements, but symbolic evaluation together with other techniques that use the program's knowledge can. One such technique could analyze a ruleset together with all the other rulesets that use it or that it uses. In the process, the optimizer constructs a database of global information concerning each ruleset that can be used in applying optimization rules.

CONCLUSIONS

ROSIE is a powerful expert system programming language. Its combination of English-like syntactic structure and intuitive semantics makes it a powerful prototyping and development environment for knowledge engineers and domain experts, both of whom can use it to communicate with one another and with the computer.

ROSIE has shown that syntactic parsing techniques can go a long way toward humanizing programming languages. Syntactic resolution of word usage seems sufficient for encoding the prescriptive knowledge of a domain. Admittedly, domain semantics—the meaning ascribed to words within a domain—are very important to understanding ideas

within that domain, but a knowledge engineer needs only to define them as completely as needed.

ROSIE's major drawback, its slow execution speed, remains a constraint on broader use. We are attacking this problem directly, and we expect that future versions of ROSIE will not only be expressive, readable, and writable, but will also support development and execution of expert systems in reasonable time frames.

REFERENCES

Anderson, R. H., M. Gallegos, J. J. Gillogly, R. B. Greenberg, and R. Villanueva, *RITA Reference Manual*, The Rand Corporation, R-1808-ARPA, September 1977.

Anderson, R. H., and J. J. Gillogly, *Rand Intelligent Terminal Agent (RITA): Design Philosophy*, The Rand Corporation, R-1809-ARPA, February 1976.

Beebe, H. M., H. S. Goodman, G. L. Henry, and D. S. Snell, "The Adept Workstation: A Knowledge Based System for Combat Intelligence Analysis," *Proceedings of the 7th MIT/ONR Workshop on C3 Systems*, MIT, Cambridge, Massachusetts, 1984.

Callero. M., D. A. Waterman, and J. R. Kipps, *TATR: A Prototype Expert System for Tactical Air Targeting*, The Rand Corporation, R-3096-ARPA, August 1984.

Erman, L. D., and V. R. Lesser, "A Multi-Level Organization for Problem Solving Using Many Diverse Cooperating Sources of Knowledge," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, USSR, 1975.

Fain, J., D. Gorlin, F. Hayes-Roth, S. Rosenschein, H. Sowizral, and D. Waterman, *The ROSIE Language Reference Manual*, The Rand Corporation, N-1647-ARPA, December 1981.

Fain, J., F. Hayes-Roth, H. Sowizral, and D. Waterman, *Programming in ROSIE: An Introduction by Means of Examples*, The Rand Corporation, N-1646-ARPA, February 1982.

Hayes-Roth, F., D. Gorlin, S. Rosenschein, H. Sowizral, and D. Waterman, *Rationale and Motivation for ROSIE*, The Rand Corporation, N-1648-ARPA, November 1981.

McArthur, D., P. Klahr, and S. Narain, *ROSS: An Object-Oriented Language for Constructing Simulations*, The Rand Corporation, R-3160-AF, December 1984.

Quinlan, J. R., *INFERNO: A Cautious Approach to Uncertain Inference*, The Rand Corporation, N-1898-RC, September 1982.

Schwabe, W., and L. M. Jamison, *A Rule-Based Policy-Level Model of Nonsuperpower Behavior in Strategic Conflicts*, The Rand Corporation, R-2962-DNA, December 1982.

Shapiro, N., H. E. Hall, R. H. Anderson, and M. LaCasse, *The RAND-ABEL Programming Language: History, Rationale and Design*, The Rand Corporation, R-3274-NA, August 1985.

Sowizral, H. A., "Experiences with Distributed Heuristic Agents in ROSIE," *Proceedings of the IEEE Conference on Systems, Man, and Cybernetics*, Bombay, India, 1983.

Waterman, D. A., R. H. Anderson, F. Hayes-Roth, P. Klahr, G. Martins, and S. J. Rosenschein, *Design of a Rule-Oriented System for Implementing Expertise*, The Rand Corporation, N-1158-1-ARPA, May 1979.

Waterman, D. A., and M. A. Peterson, *Models of Legal Decisionmaking*, The Rand Corporation, R-2717-ICJ, 1981.